User Manual for MCK. – DRAFT Version 0.0.3

Peter Gammie and Ron van der Meyden

June 18, 2003

Contents

Chapter 1

Introduction

This manual explains how to use MCK, a prototype model checker for temporal and knowledge specifications. It assumes some familiarity with the idea of model checking [4, 5, 9], and temporal and epistemic logics [7], but is otherwise self-contained.

This chapter introduces the general scenario to which the MCK system may be applied (Section 1.1) and describes an abstract model that underlies the system (Section 1.2). The semantics of the constructs of the MCK systems is most easily understood with repsect to this model. The MCK system itself uses a more concrete syntax designed to facilitate the encoding of examples. The remainder of the manual describes this concrete syntax and its associated semantics. Chapter 2 discusses the language used to model a scenario in MCK. The language used to describe the specifications that MCK checks in these scenarios is discussed in Chapter 3. Chapter 4 presents a number of examples that can be analysed using the system. The description of syntax and semantics in these chapters is semi-formal; a formal abstract syntax for the inputs to MCK is presented in Appendix A, and Appendix B provides a formal operational semantics for MCK programs.

1.1 The Model Checking Scenario

The overall scenario that can be analysed using the system has the following general structure. We consider that we are modelling a situation where some number of *agents* (which might be players in a game, actors in an economic setting, or processes, programs or components in a computational setting) interact in the context of an *environment*. A *state* of the system consists of a state of the environment together with a *local state* for each of the agents. The agents have the capacity to perform certain *actions* in this environment. The effect of actions is to change the state of the system. Each of the agents performs these actions acccording to a *protocol*, or set of rules, which describes the allowable choices of the next action at each point of time. The agents have *incomplete information* about the state of the system: their possible information is limited by the fact that they are able to *observe* only part of the state at each instant of time.

The MCK system can be applied to the analysis of this type of setting by the use of *model checking* techniques. The input to the MCK system consists of a file, or set of files, that describe:

1. the environment in which the agents operate, including:
   - the possible states of the environment,
   - the initial states of the environment,
   - the names of agents operating in this environment,
   - how the agents' actions change the state of the environment,
   - (optionally) a set of *fairness* conditions, which constrain the infinitary behaviour of the system (ensuring, e.g. that some agent is not kept waiting forever for a requested event to occur);

2. the protocol by which each of the named agents chooses their sequence of actions, including:
   - the structure of local states maintained by the agent to make such choices and record other useful information,
   - the possible initial values of the agent's local states, and
   - a description of what parts of the state are observable to the agent;
3. a number of *specification formulas*, expressing some property of the way that the agent's knowledge evolves over time.

Both the possible state changes described in the environment and the agents' choices of action may be non-deterministic, which means that the system may evolve over time in a potentially large number of different ways. The output produced by the MCK system is, for each of the specification formulas, an answer to the question of whether, for the scenario modelled, the agents' knowledge is in fact guaranteed to evolve according to the specification, for every possible evolution of the system.

The MCK system currently allows several different approaches to the description of the temporal and epistemic aspects of the specification formulas. In the epistemic dimension, agents may use their observations in a variety of ways to determine what they know. One way (the *observational* interpretation of knowledge) is to make inferences about the state based just on their latest observation. Another way (the *clock* interpretation of knowledge), permitting more information to be extracted, is to compute knowledge using both the current observation and the current clock value. Finally, even more information can be extracted by the agent if it uses a complete record of all its observations to date to determine what it knows (this is called the *synchronous perfect recall* interpretation of knowledge). In the temporal dimension, the specification formulas may describe the evolution of the system along a single computation (i.e. using linear time temporal logic, or LTL), or it may describe the branching structure of all possible computations (i.e., using the branching time, or *computation tree logic* CTL). The system currently supports different combinations of all these parameters to different degrees: in some cases this is because the implementation remains to be undertaken, in others because there are inherent computational reasons why the problem is difficult or impossible to implement.

Figure 1.1 presents an example of the input file to the system, modelling a scenario in which there is a single agent in the environment, a robot called `Robot` (running the protocol `"robot"`) operating in an environment consisting of 8 possible positions, and sensing the position using a noisy sensor, whose values are recorded in the variable `sensor`, which is observable to the agent. The example contains a single specification formula, indicated by the construct **spec_obs_ltl**, which indicates the formula uses linear time temporal logic operators and that the knowledge operator `Knows` is to be interpreted using the observational interpretation for knowledge. A more elaborate version of this example is discussed at greater length in section 4.1.

In addition to determining whether a specification formula is true or false in a given scenario, it is intended that future versions of the system will provide additional forms of support for the analysis of such scenarios, such as permitting the user to check the model by navigating through executions of the scenario, and presenting counterexamples when specification formulas are found to be false.

1.2 Background Theory

This section describes an abstract mathematical model that underlies the MCK system. It is closely related to models used in works including [12, 13, 14], which present some of the algorithms and data structures that underlie the analysis performed by the MCK system. (These papers themselves use a variety of formal modellings, and the differences between these papers and the model described here is largely a matter of mathematical presentation.) The remainder of the manual provides a more concrete syntax and semantics for the model developed here.

We first present an abstract view of the semantics of MCK programs that is adequate from the point of view of the main constructs of the specification language used in MCK. From this perspective, as system comprised of a set of interacting agents is, at each point of time, in some *global state*. Write $\mathcal{G}$ for the set of all possible global states of the system. A *run* is a possible history of such states, and can be modelled by an infinite sequence $r = s_0, s_1, s_2, \ldots$ where each $s_k \in \mathcal{G}$. We write $r(m)$ for the m-th state in this sequence.

The behaviour of a system is typically non-deterministic, as agents may have a choice of what actions to perform at any given point of time, and the environment may also model non-deterministic events such as communication links failures and delays. We may model this non-determinacy by representing the system as a set $\mathcal{R}$ of runs, intuitively all the possible ways that the history of the system may evolve.

In general, agents are not able to observe the entire state of the system. We model this by means of a function $O_i : \mathcal{G} \rightarrow \mathcal{O}_i$, for each agent i where $\mathcal{O}_i$ is the set of observations made by agent i. Intuitively, $O_i(s)$ is the information that is visible to agent i when the system is in the global state s. Based on their observations, agents are able to make inferences about the situation in which they find themselves, i.e., the particular state, time and past and future history. We model such a situation as a *point*, represented as a pair (r, m), where r is a run and $m \in \mathbf{N}$ is a time.

In order to determine what they know, agents may make use of their observations in a variety of ways. We capture the specific way that the agents use their observations for purposes of computing knowledge by assigning them a *local state* with respect to a *view* at each point of the system. We write $r_i^x(m)$ for the local state of agent i at the point (r, m), where x is the view. The simplest view is the observational view `obs`, where the agent uses just its current observation to determine what it knows. The local state in this case is defined by $r_i^{\texttt{obs}}(m) = O_i(r(m))$. Somewhat more informative to the agent is the *clock view* `clock`, defined by $r_i^{\texttt{clock}}(m) = (m, O_i(r(m)))$. Here the agent uses both its current observation and the current global clock value to determine what it knows. Most informative is the *synchronous perfect recall view*, defined by $r_i^{\texttt{spr}}(m) = \langle O_i(r(0)), \ldots, O_i(r(m)) \rangle$. Here the agent uses its complete sequence of observations to the current time to determine what it knows.

For each view x, we may define a relation of *indistinguishability* on points: two points (r, m) and (r', m') are said to be indistinghuishable to agent i, written $(r, m) \sim_i^x (r', m')$ if the agent has the same local state with respect to the view at those two points, i.e. if $r_i^x(m) = (r')_i^x(m')$. Intuitively, the set of all points that are indistinguishable from a point (r, m) is the set of all points that the agent considers to be possibly the current point, when using the information capture in that view. We may therefore say that agent i *knows* (with respect to view x) that a formula ϕ holds at a point (r, m) if ϕ holds at all points (r', m') such that $(r', m') \sim_i^x (r, m)$. In MCK, the statement that agent i knows ϕ is written as `Knows` i ϕ, with the view indicated at the level of the larger formula of which this statement is a part.

The above definitions suffice to give semantics to the specification langauge of MCK. (In addition to the knowledge operator just defined, there is a variety of temporal logic operators. These are defined in Chapter 3.) In order to enable the user to describe the system in which a formula is to be checked, MCK provides a systems modelling language that consists of two parts. The set of runs of a system is taken to be generated by the agents each running a *protocol* by which they choose actions available to them in a given *environment*. The global states $\mathcal{G}$ are made up of two components: a state of the environment and a protocol state for each of the agents.

Abstractly, we model the environment as a finite-state transition system, with the transitions labelled by the agents' actions. For each agent $i = 1 \ldots n$ let A_i be a set of *actions* associated with agent i. A *joint action* consist of an action for each agent, i.e., the set of joint actions is the cartesian product $A = A_1 \times \ldots \times A_n$. Define a *finite environment for n agents* to be a tuple $\mathcal{E}$ of the form $\langle S_e, I_e, \tau, \langle \alpha_{f_0}, ..., \alpha_{f_n} \rangle \rangle$ where the components are as follows:

1. S_e is a finite set of *states of the environment*. (Concretely, these are given in MCK by specifying a set of typed variables.)

2. I_e is a subset of S_e, representing the possible *initial states* of the environment. (Concretely,

this is specified in MCK by a constraint on the environment variables.)

3. τ_e is a function mapping each joint action $\mathbf{a} \in A$ to a nondeterministic state transition function $\tau(\mathbf{a}) : S_e \rightarrow \mathcal{P}(S_e)$. Intuitively, when the joint action $\mathbf{a}$ is performed in the state s, the resulting state of the environment is one of the states in $\tau(\mathbf{a})(s)$. (Concretely, this is given in MCK by writing a nondeterministic program that computes this state transition function.)

4. each α_i in $\langle \alpha_{f_0}, ..., \alpha_{f_n} \rangle$ is a subset of S_e, used to model a *fairness* condition. Intuitively, for each i, there is a state $s_i \in \alpha_i$ that occurs infinitely often in every run.

The behaviour of agents is given concretely in MCK by writing a program that describes their choice of action at each point of time. Each such program defines a set of states S_i, given concretely by means of a set of variables, which includes the *program counter*, a special variable whose value is the position in the program at which control resides at the given point in time. Together, the states of the environment and these protocol states determine the set of global states $\mathcal{G}$: these consist of tuples $\langle s_e, s_1, \ldots, s_n \rangle$ comprised of a state s_e in S_e and a state s_i in S_i for each agent i.

Abstractly, agent i's program defines not just the protocol states S_i, but in fact a tuple $\langle S_i, I_i, P_i, \mu_i \rangle$, where

1. I_i is the set of possible initial states of the protocol. Concretely, this is given by means of a constraint on the program variables.

2. $P_i : \mathcal{G} \rightarrow \mathcal{P}(A_i)$ is a function mapping global states to a set of possible actions for the agent.

3. $\mu_i : \mathcal{G} \times A_i \rightarrow \mathcal{P}(S_i)$ is a function that describes how the protocol state (including the program counter) is updated when the agent performs an action.

We remark that the functions P_i and mu_i depend on global states rather than just protocol states because agents may observe some of the environment variables and use these observations in deciding what to do. Note that one agent cannot directly access another agent's local variables; this is a syntactically-enfored constraint on agent programs. Similarly, the environment transition function τ cannot make use of any agent-local state.

Using these components, we may now define a *global initial state* to be a global state $\langle s_e, s_1, \ldots, s_n \rangle$ such that $s_e \in I_e$ and $s_i \in I_i$ for each agent i. Moreover, we define a *global state transition relation* T on global states as follows. If there are n agents, sTs', where $s = \langle s_e, s_1, \ldots, s_n \rangle$ and $s' = \langle s'_e, s'_1, \ldots, s'_n \rangle$ if there is a joint action $\mathbf{a} = \langle a_1, \ldots, a_n \rangle$ such that for each agent i, $a_i \in P_i(s)$ is an action that may be selected by agent i's protocol, the state $s'_e \in \tau(\mathbf{a})(s_e)$ is one of the possible outcomes of performing the joint action, and for each agent i, the state $s'_i \in \mu_i(s, a_i)$ is the result of updating the protocol state accordingly.

We may now define the set of runs generated when the agents execute their protocols in the given enviroment as the set of all runs $r = s_0, s_1, \ldots$ such that s_0 is a global initial state, and for for each $k \geq 0$, we have $s_k T s_{k+1}$, i.e, there is a transition from s_k to s_{k+1}, and the fairness conditions $\langle \alpha_{f_0}, ..., \alpha_{f_n} \rangle$ are satisfied, i.e., there exist states $s_{f_0}, ..., s_{f_n}$ where $r(m) = s_{f_i}$ for infinitely many m, and $s_{f_i} \in \alpha_{f_i}$ for $0 \leq i \leq n$.

1.3 Installation and Invocation

The program is implemented in Haskell and makes use of extensions only implemented by the Glasgow Haskell Compiler (`http://haskell.org/ghc/`). It uses David Long's Binary Decision Diagram package, available from CMU (`http://www-2.cs.cmu.edu/~modelcheck/bdd.html`), and a Haskell binding of it (that should accompany the source distribution).

See the file `INSTALL` for detailed installation instructions.

Invocation

The `mck` program accepts the following flags:

- `-c` or `--counter-examples`: Generate counter-examples (not fully implemented).
- `-d[Int]` or `--debug[=Int]`: Output BDD debugging info and stats. This is not particularly useful for end users.
- `-e` or `--environment`: Output the environment information.
- `-f` or `--formula`: Output the BDD formulas in human-readable form. This is not particularly useful for end users.
- `-o File` or `--var-order=File`: Output the variable order to a file.
- `-p` or `--protocol`: Output the pre-processed protocol information.
- `-r[s|w]` or `--reorder[=s|w]`: Enable BDD variable re-ordering. The two methods are sifting and window-based sifting (see the CMU/Long BDD manpage for details).
- `-s File` or `--set-var-order=File`: Load variable ordering from a file.

It expects the filename of an input script to be supplied.

The program will search the current directory for files matching the names of any protocols that are used in an input script but not defined there. See Section 4.2 for an example of this.

```
environment "Robot Env"

-- There are 8 positions in the world.
-- If the robot is really at position p, then the sensor will have a
-- value ∈ {p − 1, p, p + 1}, for a truncating interpretation of arithmetic.
type Pos = {0..7}

position : Pos
sensor : Pos

agent Robot "robot" ( sensor )

init_cond = position == 0 /\ sensor == 0

-- At each time step, the environment moves the robot one step to the
-- right, and generates a new sensor reading.
resolve
begin
  if neg Robot.Halt -> position := position + 1
  fi;
  if True -> sensor := position − 1
  [] True -> sensor := position
  [] True -> sensor := position + 1
  fi
end

spec_obs_ltl = G (sensor >= 3 <-> Knows Robot position in {2..4})

-- The “bike handbrake” protocol
-- In order to stop moving, the robot needs to keep the brake pressed.
protocol "robot" (sensor : observable Pos)

begin
  do neg (sensor >= 3) -> skip
  [] break -> <<Halt>>
  od;
  while True do halted :: <<Halt>>
end
```

Figure 1.1: A simplified version of the robot example.

Chapter 2

The Input Language

This chapter provides an informal description of the language used to describe the model checking scenario (Section 1.1), which is more formally described in Appendix A (syntax) and Appendix B (semantics).

An input script consists of a bunch of environment declarations, one or more specifications, and zero or more protocols, as illustrated in Figure 2.1. We proceed by describing the lexical conventions, the role of types, the structure of agent protocols and finally the variety of environment declarations, deferring specifications and fairness constraints to Chapter 3.

2.1 Lexical Structure

The lexical structure of the language follows Haskell [11] closely. Specifically:

Comments begin with '−−' and terminate at the end of the line they appear on.

Constants start with an upper-case letter, and can be followed by any number of a mix of alphanumeric characters and underscores. These are used in enumerated types (Section 2.2), actions and agent names.

Variables start with a lower-case letter followed by any number of a mix of alphanumeric characters and underscores. Program variables and labels belong to this class.

Relational variables start with an underscore followed by any number of a mix of alphanumeric characters and underscores. These are used in μ-calculus specifications (Section 3.3).

2.1.1 Reserved Words

The reserved words in the MCK input language, of which there are many, are spelt out in the formal syntax given in Appendix A.

2.2 Types

All types are finite ("enumerated") totally-ordered objects, and so can be used in relational and arithmetic expressions in the natural way.

Types can be introduced by either explicitly enumerating their elements, or by specifying a range of integers. The concrete syntax can be found in Section A.1. Lexically, the type name and elements are **Constants**, and elements do not have to have a unique type. Note that this *ad hoc* overloading of constants restricts the allowable structure of expressions (see Sections 2.3.1 and A.3).

The canonical ordering on the elements is the textual order in which they are defined. For example, in the context of the declarations:

```
type Int3 = {0..7}

x : Int3
y : Int3
...
init_cond = x == 3 /\ y == 4
```

we have $x < y$ in all initial states.

The only primitive pre-defined type is *Bool*, which behaves as if it were defined like so:

$$\textbf{type}\ Bool = \{False, True\}$$

Arrays

Environment variables can be given an array type using the standard C syntax. Their main utility is in abstracting a protocol from the concrete number of agents present in a given scenario. See Section 4.2 for an example of where this is useful.

2.3 Agent Protocols

A protocol defines an agent's behaviour, and as such is a program written in an imperative language reminiscent of Dijkstra's *guarded commands* [3]. The basic structure of such definitions is illustrated in Figure 2.1, and is more formally spelt out in Section A.2.

In essence, a protocol has a *name*, a list of *environment parameters*, a list of *local variables*, an *initial condition*, and a *statement block*. As such, it looks like a C- or Pascal-style function definition, and indeed it behaves as if the environment parameters are passed by value. This is spelt out in more detail in the following section on expressions.

The local variables are simply identifier-type pairs, and the initial condition is either a boolean expression mentioning only local variables or the special form **all_init** which indicates that all variables should be initialised to the first value in their type – which is $False$ for boolean variables.

2.3.1 Expressions

The expression sub-language is used to define conditions in the **if** and **do** constructs, and the right-hand-sides of assignment statements. The full syntax is spelt out in Section A.3, and what follows is an overview.

The only operators that are fully recursively general are the *boolean* constructs /\ (and), \/ (or), $->$ (implication), $<->$ (bi-implication) and *xor*. All other operators are used to either construct propositions or basic arithmetic formulas.

Expressions can mention any subset of the local variables and environment parameters that are flagged **observable** (Section 1.2 explains what the **observable** attribute means, and Figure 3.2 gives an example of their use in expressions).

In the context of the definitions in Section 2.2 involving $Int3$ and $x : Int3$ with value 3, basic expressions of the following forms are valid, and have the specified value:

Operator	Examples
Enumeration	x **in** $\{1,2,4\} \Rightarrow False$, x **in** $\{3..7\} \Rightarrow True$.
Equality	$x == 2 \Rightarrow False$, $x/ = 3 \Rightarrow False$.
Relational	$x > 3 \Rightarrow False, x >= 3 \Rightarrow True, x < 3 \Rightarrow False, x <= 3 \Rightarrow True.$
Truncating Arithmetic	$x + 1 \Rightarrow 4, x - 4 \Rightarrow 0.$

It is envisaged that other forms of arithmetic will be useful, and these will be implemented in the future.

2.3.2 Well-typed expressions

As all types are enumerated, we have very liberal overloading rules. Indeed, the type checker simply computes an approximation to what values an expression can take on and ensures it's a subset of **Bool**, in the case of boolean formulas, or of the type of the variable being assigned to otherwise.

In contrast to (for example) [10], there is enough information in each subexpression to uniquely determine it's type[1] and so no backtracking is required.

2.3.3 Statements

The imperative language is based on Dijkstra's *guarded commands* [3]. The only substantial deviation from his presentation is the addition of a **break** branch in a **do** statement which is executed when the **do** loop terminates (i.e. when all other conditions are false). Why this is important is discussed in more detail in Sections 4.1 and Appendix B.

Core language

Alternatives: The non-deterministic choice statement has the form:

$$\mathbf{if}\ cond_1 \rightarrow C_1\ ...\ []\ cond_i \rightarrow C_i\ ...\ [\mathbf{otherwise} \rightarrow C_o]\ \mathbf{fi}$$

where each command C_i is eligible for execution only if the corresponding condition $cond_i$ evaluates to true in the current state. If, for all i, $cond_i$ evaluates to false, then C_o is executed. If the **otherwise** branch is absent then an implicit **otherwise** → **skip** is introduced.

This construct consumes no time steps.

Repetition: The non-deterministic iteration statement has the form:

$$\mathbf{do}\ cond_1 \rightarrow C_1\ ...\ []\ cond_i \rightarrow C_i\ ...\ [\mathbf{otherwise} \rightarrow C_o]\ [\mathbf{break} \rightarrow\ C_b\ ]\mathbf{od}$$

where each command C_i is eligible for execution only if it's corresponding condition $cond_i$ evaluates to true in the current state, a process which is repeated until all conditions evaluate to false. At this time the C_b statement is executed if the **break** branch is present; otherwise the system implicitly executes the **skip** command.

This construct consumes no time steps.

Sequential Composition: An arbitrary number of statements $C_1, ..., C_n$ to be executed in sequence can be aggregated by writing: **begin** $C_1; ...; C_n$ **end**.

This construct consumes no time steps.

Actions: An *action* captures the essence of how an agent can contribute to a state update: it both sends a signal to the environment and specifies how the local state of the agent should change. This *select/resolve* model is detailed in the introduction (Chapter 1), and the corresponding environment declarations are described in Section 2.4.5.

An action has one of two forms:

$$<<\ Action\ >>$$
$$<<\ Action\ |\ var_1 := expr_1;\ ...\ ; var_n := expr_n\ >>$$

Note that $var_1, ..., var_n$ must be distinct local variables.

In the future, it is expected that the implementation will support actions parameterised by variables and constants. In the interim, *shared variables* primitives are provided as described in Section 2.4.2.

An action costs 1 time step to execute.

[1]With the exception of bare constants.

Derived forms

This section gives the expansion $[\cdot]$ of the various derived forms in terms of the core language.

skip: The do-nothing statement that consumes 1 time step. The expansion is as follows:

$$[skip] = << \mathit{NillAction} >>$$

assignment: A protocol can assign the value of an expression to a local variable in 1 time step. The expansion is as follows:

$$[var := expr] = << \mathit{NillAction} \mid var := expr >>.$$

Multiple variables can be assigned to in the same time step:

$$[<< var_1 := expr_1; ...; var_n := expr_n >>] = << \mathit{NillAction} \mid var_1 := expr_1; ...; var_n := expr_n >>.$$

This is also termed an *atomic* or *parallel* assignment.

if-then-else: A choice construct derived from **if**. The expansion is as follows:

$$[\mathbf{if}\ cond\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2] = \mathbf{if}\ cond\ \rightarrow\ C_1\ []\ \mathbf{neg}\ cond\ \rightarrow\ C_2\ \mathbf{fi}$$

while: A repetition construct derived from **do**. The expansion is as follows:

$$[\mathbf{while}\ cond\ \mathbf{do}\ C] = \mathbf{do}\ cond\ \rightarrow\ C\ []\ \mathbf{break} \rightarrow\ \mathbf{skip}\ \mathbf{od}$$

Intuitively, C is executed until *cond* becomes false. The **break** → **skip** branch means that it takes 1 time step to exit the while loop.

2.3.4 Labels

All statements can be given a not-necessarily-unique label, which is written like so:

$$label :: C$$

where *label* belongs to the **Variable** lexical class.

Labels may only be used within specifications, and are further discussed in Section 3.7.

2.4 The Environment

This section describes the environment declarations that specify how the agents communicate and which protocol they execute. Discussion of the other declarations appearing under the **environment** section header (specifications and fairness constraints) are deferred to Chapter 3.

Note that the declarations must appear in the script in the same order as they are presented here.

2.4.1 Types

The first set of declarations introduce zero or more types, as specified in Section 2.2.

2.4.2 Shared Variables

The only communication medium currently implemented is a shared variables abstraction. In essence, such a variable has a particular value that persists, unless it is written to. The model checker resolves concurrent reads and writes to a variable by exploring all possible interleavings of these actions.

Shared variables are declared in the same fashion as agent-local variables.

There are two specialised forms of actions used in protocols to access the shared variables. The first:

$$<< local\_var \ := \ environment\_var.read() >>$$
$$<< local\_var \ := \ environment\_var.read() \mid var_1 := expr_1; \ \dots ; var_n := expr_n >>$$

assigns the value of an environment variable to a local variable, taking into account concurrent writes. Note that the environment variable need not be observable in this case (in constrast to Section 2.3.1).

The second form:

$$<< environment\_var.write(expr) >>$$
$$<< environment\_var.write(expr) \mid var_1 := expr_1; \ \dots ; var_n := expr_n >>$$

provides a way for an agent to assign a value to a shared variable.

2.4.3 Agent Bindings

The next set of declarations bind distinct agent names to the protocols they run, and instantiate each protocol's environment parameters. There must be at least one agent in the system.

Note that this binding mechanism introduces the possibility of *aliasing* – having multiple names for the same variable. This is not an issue as the usual problem of interference doesn't arise – agents can only write to a single environment variable per time step, and conflicting writes are resolved at the memory-location level.

2.4.4 Initial conditions

If an **init_cond** declaration is present, then the environment variables are constrained to satisfy it in the first state of the system. Every such solution leads to a distinct initial state.

2.4.5 Resolution

An alternative communication mechanism is to provide a **resolve** clause, which implements the second part of the *select/resolve* model described in the introduction (Chapter 1).

In essence, this clause is identical to a statement block in a protocol definition, except that:

- Only non-looping constructs are allowed (no **do** or **while** loops).
- Guards are in terms of agent-qualified actions and environment variables, and are of a constrained form (see Appendix A).

Informally, this statement block is executed after the agents have decided which action they wish to perform, and a new state is generated via their joint action.

At the moment only actions without arguments are implemented.

This mechanism is used in the robot example of Section 4.1.

```
environment "environment name"

-- Types. (zero or more)
type TypeName0 = { ... elements ... }
...
type TypeNameT = { ... elements ... }

-- Shared variables. (zero or more)
varDec0 : Type
...
varDecN : Type

-- Agent bindings. (at least one)
agent AgentName1 "protocol for agent 1" ( ... env variables ... )
...
agent AgentNameM "protocol for agent M" ( ... env variables ... )

-- Environment initial conditions. (optional)
init_cond = ... boolean expression involving env variables ...

-- Resolve clause. (optional)
resolve
begin
  ... statements ...
end

-- Specifications. (at least one)
<specification_type> = ... temporal and knowledge formula ...

-- Fairness constraints. (zero or more)
fairness = ... CTL formula ...

-- Protocol declarations. (zero or more - can be in a separate file)
protocol "first protocol name" ( ... env parameters ... )
localVar0 : Type
...
localVarK : Type
  where ... boolean expression involving local variables ...

begin
  ... statements ...
end
```

Figure 2.1: The structure of an input script.

Chapter 3

The Specification Language

There are two dimensions to the specification language: the temporal logic semantics (leading $\mathtt{X}^n$/LTL/CTL) and the knowledge semantics (observational, clock, perfect recall). The combinations of the two that are implemented are shown in Figure 3.1, and the various operators are described in the following sections.

Note that the "leading $\mathtt{X}^n$" semantics simply signifies taking n steps before evaluating the rest of the formula, which must not contain temporal operators.

3.1 The Propositional Core

Basic propositions in this language can be formed in several ways:

Boolean variables can be used directly: environment variables are denoted by their names, and agent variables can be accessed as $AgentName.variable$. Note that labels also use this syntax; see Section 3.7 for details.

Equality between a variable of arbitrary type and an element of that type is denoted as $var ==$ $Constant$. The negation of this is written $variable/ = Constant$.

Relational expressions between variables and constants are permissible; for example $var > 3$, $4 < var$ and var **in** $\{3, 5, 7\}$ are all valid propositions, provided var is given a type that has (at least) $\{3, 4, 5, 7\}$ as elements.

The usual boolean connectives (Section A.5) can be used to combine propositions. Note the absence of arithmetic operations; the form in which they can appear in expressions (Section 2.3.1) is significantly less useful in specifications.

	Observational	Clock	Perfect Recall
leading $\mathtt{X}^n$	**spec_obs**	**spec_clock**	**spec_pr**
CTL	**spec_obs**		
LTL	**spec_obs_ltl**		

Figure 3.1: The combinations of temporal and knowledge semantics currently implemented in the model checker.

3.2 Computation Tree Logic

Computation Tree Logic is a well-known *branching-time* logic used (for example) in SMV [5]. The available operators and an informal semantics are as follows:

Operator	Description
$\mathbf{AX}\ f$	f in all next states.
$\mathbf{EX}\ f$	f in at least one next state.
$\mathbf{A}[f\ \mathbf{U}\ g]$	on all paths, f until g.
$\mathbf{E}[f\ \mathbf{U}\ g]$	on at least one path, f until g.
$\mathbf{AF}\ f$	On all paths, in some future state, f.
$\mathbf{EF}\ f$	On at least one path, in some future state, f.
$\mathbf{AG}\ f$	On all paths, in all future states, f.
$\mathbf{EG}\ f$	On at least one path, in all future states, f.

An example appears in Figure 3.2. In English, the specification might be rendered as "in all reachable states, there exists a future state where *A.var* is the case". The set of reachable states is subject to fairness constraints (see below).

3.3 μ-calculus Constructs

The μ-calculus adds greatest and least fixed-points to the branching-time model that CTL uses, and can indeed express all (unfair) CTL constructs. Concretely, in **spec_obs** specifications the following two operators can be used:

gfp $\_Z\ f$ greatest fixed point of f wrt variable $\_Z$.

lfp $\_Z\ f$ least fixed point of f wrt variable $\_Z$.

with the constraints that all *relational variables* (such as $\_Z$) used in f fall under an even number of negations, and that the specification as a whole is closed with respect to relational variables.

For example, the (unfair) CTL operator $\mathbf{EG}f$ can be written as **lfp** $\_Z\ (f \vee \mathbf{AX}\ \_Z)$.

Note that the evaluation of fixpoints is quite naive at the moment as the Emerson-Lei algorithm (or a more-recent refinement) isn't implemented. See [5, Chapter 7] for details.

3.4 Linear Temporal Logic

Linear Temporal Logic is a well-known *linear-time* logic used (for example) in SPIN [8]. Informally, the available operators and semantics are as follows:

Operator	Description
$\mathbf{F}\ f$	eventually f.
$\mathbf{G}\ f$	always f.
$f\ \mathbf{U}\ g$	f until g.
$\mathbf{X}\ f$	f in the next state.
$\mathbf{X}$ `int` f	f in `int` steps.

LTL can directly encode fairness conditions, but it is also possible to use CTL fairness constraints (Section 3.6).

An example of using LTL specifications is the following simple, not-fully-correct mutual exclusion algorithm.

---

environment `"mutex"`

turn1 : Bool

```
turn2 : Bool

agent M1 "mutex" ( turn1, turn2 )
agent M2 "mutex" ( turn2, turn1 )

-- Non-deterministic choice of who goes first.
init_cond = turn1 xor turn2

-- Safety
spec_obs_ltl = G neg (M1.in_cs /\ M2.in_cs)

-- Non-blocking FIXME
spec_obs_ltl = G (((F M1.left_cs) /\ neg M1.in_cs) -> F M1.trying)
spec_obs_ltl = G (((F M2.left_cs) /\ neg M2.in_cs) -> F M2.trying)

-- Non-strict sequencing (fails)
spec_obs = EF (M1.in_cs /\ E[M1.in_cs
                          U (neg M1.in_cs /\ E[neg M2.in_cs U M1.in_cs])])

-- Liveness (need to consider fairness)
spec_obs_ltl = G (((F M2.left_cs) /\ M1.trying) -> F M1.in_cs)
spec_obs_ltl = G (((F M1.left_cs) /\ M2.trying) -> F M2.in_cs)

protocol "mutex" ( env_turn1 : observable Bool, env_turn2 : Bool )
in_cs : Bool
  where neg in_cs

begin
  while True do
  begin
    while env_turn1 /= True do trying :: skip;
    -- Critical section
    in_cs := True;
    do in_cs -> skip
    [] in_cs -> in_cs := False
    [] break -> skip
    od;
    -- End critical section
    left_cs :: << env_turn1.write(False) >>;
    << env_turn2.write(True) >>
  end
end
```

3.5 Knowledge Modalities

The knowledge modality is written **Knows** $Agent\ formula$. The various semantics for this operator were spelt out in Section 1.2.

For the clock and observational semantics, a *common knowledge* operator is also available, written **CK** $\{Agent_1, ..., Agent_n\}$ $formula$ ($formula$ is common knowledge to the specified agents), or **CK** $formula$ ($formula$ is common knowledge to all agents).

3.6 Fairness Constraints

MCK supports fairness constraints in the manner described in [4, Section 7] and [5, Section 6.3]. Briefly, a fairness constraint filters the runs of the system by accepting only those along which a given CTL formula is satisfied infinitely often. This is explained in more detail in Chapter 1.

An example of using a fairness declaration to eliminate undesired runs is shown in Figure 3.2. The constraint **fairness** = $A.var$ ensures that the runs that, after some finite period of time, forevermore have var false are eliminated.

```
environment "env"

agent A "while" ()

spec_obs = AG AF A.var

fairness = A.var

protocol "while" ()
var : Bool
  where neg var

begin
  do True -> var := False
  [] True -> var := True
  od
end
```

Figure 3.2: An example of a CTL specification.

While it is tempting to try to use labels to specify which branches of **if** and **while** should be treated fairly, it is not as straightfoward as one might hope. The following section gives details.

3.7 The Use of Labels

Explication of the semantics of labels requires a distinction to be drawn between state transitions that are *enabled* versus those which are *taken*. For example, when control reaches the following program fragment:

```
if
   True -> l0 :: var := 0;
[] True -> l1 :: var := 1;
[] True -> l2 :: var := 2;
fi
```

we have that all branches of the **if** statement are enabled, but only one can be (non-deterministically) taken.

Concretely, the proposition $AgentName.label$ evaluates to $True$ if and only if a statement labelled by *label* is enabled. If there are several statements with the same label, then the proposition is true if any of them are enabled.

The biggest trap with this arrangement is that it is not always adequate for specifying fairness constraints. Consider, for example, the constraint that the middle branch is executed infinitely often. It is tempting to write the fairness constraint like so:

$$fairness = AgentName.l1$$

but this merely asserts that the branch is enabled infinitely often, and does not rule out runs where it is never actually taken. The simplest solution is to add an auxiliary variable that tracks when a branch is taken, and re-formulate the constraint in terms of it:

$$fairness = var == 1$$

Chapter 4

Examples

This chapter illustrates the kinds of properties the model checker can verify, and some of the subtleties that may arise when formalising a system.

4.1 The Robot example

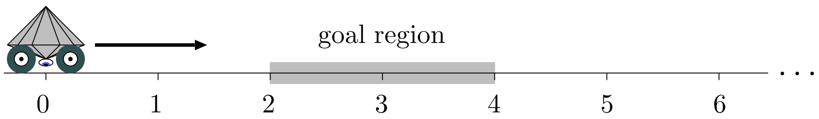

Figure 4.1: Autonomous Robot

This example is taken from [1]. To quote the summation found in [6]:

> A robot travels along an endless corridor, which in this example is identified with the natural numbers. The robot starts at 0 and has the goal of stopping in the goal region $\{2, 3, 4\}$. To judge when to stop the robot has a sensor that reads the current position. (See Figure 4.1.) Unfortunately, the sensor is inaccurate; the readings may be wrong by at most 1. The only action the robot can actively take is halting, the effect of which is instantaneous stopping. Unless this action is taken, the robot may move by steps of length 1 to higher numbers. Unless it has taken its halting action, it is beyond it control whether it moves in a step.

A sound and complete solution to this problem is to do nothing while the sensor has a value of less than 3, and halt as soon as it takes on a value of 3 or more. (The naive solution of halting iff the sensor reads 3 is sound but not complete.)

In order to model check an implementation of the robot's control policy, we need to restrict the environment to a finite number of locations. We have arbitrarily chosen to have 8 distinct locations, but any number greater than 4 is sufficient.

An input script implementing such a policy in such an environment is shown in Figure 4.2.

Note that timing is critical in this example: the robot must have continuous control over the emitted Actions, and must be able to register it's intention to halt with the environment *before* the environment decides to move it any further. These two constraints mean that using a **while** construct, with the implied **skip**-on-exit, is not sufficient for correctness. This "timing gap" is illustrated via an alternative, incorrect protocol shown in Figure 4.3. The sequence of $(position, sensor, robot\ action)$ values:

$$\langle(0,0,Nill),(1,0,Nill),(2,1,Nill),(3,2,Nill),(\mathbf{4},\mathbf{3},\mathbf{Nill}),(\mathbf{5},\mathbf{4},\mathbf{Halt}),...\rangle$$

is an example of a run where the robot decides it wants to stop when it receives a sensor reading of 3, but doesn't manage to assert the halted action until it has moved past the goal region.

4.2 The Dining Cryptographers

The problem solved by this protocol is framed as follows [2]:

> Three cryptographers are sitting down to dinner at their favorite three-star restuarant. Their waiter informs them that arrangements have been made with the maitre d'hotel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been the NSA (US National Security Agency). The three cryptographers respect each other's right to make an anonymous payment, but they wonder if the NSA is paying.

Assuming that at most one cryptographer is paying, the protocol shown in Figure 4.4 will allow all cryptographers to discover whether the NSA or one of their fellows is paying.

The details of the model checking of this protocol are given at length in [14].

Prosaically, this protocol illustrates the utility of arrays of environment variables – in this case simply to implement a broadcast. Unfortunately, the environments become increasingly baroque to define as the number of agents increases, due mainly to a blow-out in the length of the initial condition, and so it is not a complete solution to this problem.

```
environment "Robot Env"

-- There are 8 positions in the world.
-- If the robot is really at position p, then the sensor will have a
-- value ∈ {p − 1, p, p + 1}, for a truncating interpretation of arithmetic.
type Pos = {0..7}

incpos : Bool
position : Pos
sensor : Pos

agent Robot "robot" ( sensor )

init_cond = incpos /\ position == 0 /\ sensor == 0

-- At each time step the environment might move the robot one step to the
-- right, and always generates a new sensor reading.
resolve
begin
  if neg Robot.Halt ->
    begin
      position := position;
      incpos := False
    end
  [] neg Robot.Halt ->
    begin
      position := position + 1;
      incpos := True
    end
  fi;
  if True -> sensor := position − 1
  [] True -> sensor := position
  [] True -> sensor := position + 1
  fi
end

-- Knowledge-based program specification agrees with the implementation.
spec_obs_ltl = G (sensor >= 3 <-> Knows Robot position in {2..4})

-- Rule out the traces where the environment stops trying to advance.
fairness = incpos

-- The "bike handbrake" protocol.
-- In order to stop moving, the robot needs to keep the brake pressed.
protocol "robot" (sensor : observable Pos)

begin
  do neg (sensor >= 3) -> skip
  [] break -> <<Halt>>
  od;
  while True do <<Halt>>
end
```

Figure 4.2: The robot input script.

```
protocol "robotbroken-agent-protocol.mck" (sensor : observable Pos)

begin
  while neg (sensor >= 3) do skip;
  while True do halted :: <<Halt>>
end
```

Figure 4.3: The modified, incorrect robot protocol.

```
protocol "dc-agent-protocol.mck"
(
  env_paid : constant Bool,
  chan_left : Bool,
  chan_right : Bool,
  said : observable Bool[ ] -- the broadcast variables.
)

coin_left : Bool
coin_right : Bool
paid : Bool
  where all_init

begin
  -- The enviroment tells us whether we paid or not.
  << paid := env_paid.read() >>;
  -- The agent decides the coin toss to the right.
  if True -> coin_right := True
  [] True -> coin_right := False
  fi;
  << chan_right.write(coin_right) >>;
  << coin_left := chan_left.read() >>;
  << said[self].write(coin_left xor coin_right xor paid) >>
end
```

Figure 4.4: The Dining Cryptographers protocol (file `dc-agent-protocol.mck`).

```
environment "dining_cryptographers"

paid : constant Bool[3]
chan : Bool[3]
said : Bool[3]

-- Agents are numbered in the order they appear.
agent C1 "dc-agent-protocol.mck" (paid[0], chan[0], chan[1], said)
agent C2 "dc-agent-protocol.mck" (paid[1], chan[1], chan[2], said)
agent C3 "dc-agent-protocol.mck" (paid[2], chan[2], chan[0], said)

init_cond = ((neg paid[0]) /\ (neg paid[1]) /\ (neg paid[2]))
         \/ ((paid[0])     /\ (neg paid[1]) /\ (neg paid[2]))
         \/ ((neg paid[0]) /\ (paid[1])     /\ (neg paid[2]))
         \/ ((neg paid[0]) /\ (neg paid[1]) /\ (paid[2]))

-- This talks about the knowledge of the first agent.
spec_pr = X 6
            (neg paid[0]) -> ((Knows C1 (neg paid[0])
                                 /\ (neg paid[1])
                                 /\ (neg paid[2]))
                          \/ (Knows C1 (paid[1] \/ paid[2])
                               /\ (neg (Knows C1 paid[1]))
                               /\ (neg (Knows C1 paid[2]))))
```

Figure 4.5: The Dining Cryptographers environment (file `dcenv.mck`).

Bibliography

[1] Ronen I. Brafman, Jean-Claude Latombe, Yoram Moses, and Yoav Shoham. Applications of a logic of knowledge to motion planning under uncertainty. *JACM*, 44(5), 1997.

[2] D. Chaum. The dining cryptographers problem: unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.

[3] Edgar W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[4] E. Clarke, O. Grumberg, and D. Long. Verification Tools for Finite State Concurrent Systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency-Reflections and Perspectives*, volume 803, pages 124–175, Noordwijkerhout, Netherlands, 1993. Springer-Verlag.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[6] Kai Engelhardt, Ron van der Meyden, and Yoram Moses. A program refinement framework supporting reasoning about knowledge and time. In Jerzy Tiuryn, editor, *Foundations of Software Science and Computation Structures*, volume 1784 of *Lecture Notes in Computer Science*, pages 114–129. Springer-Verlag, March 2000.

[7] R. Fagin, J. Y. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge*. The MIT Press, 1995.

[8] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[9] M. Huth and M. Ryan. *Logic in Computer Science: modelling and reasoning about systems*. Cambridge University Press, 2000.

[10] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, August 1984.

[11] Simon Peyton Jones, John Huges, et al. Haskell 98: A non-strict, purely functional language. `http://haskell.org/definition/`, February 1999.

[12] Ron van der Meyden. Knowledge-based programs: on the complexity of perfect recall in finite environments (extended abstract). In *Proceedings of the Conference on Theoretical Aspects of Reasoning about Knowledge*, Renesse, Netherlands, mar 1996.

[13] Ron van der Meyden. Common knowledge and update in finite environments. *Information and Computation*, 140(2), 1998.

[14] Ron van der Meyden and Kaile Su. Symbolic model checking the knowledge of the dining cryptographers. As yet unpublished., 2002.

Appendix A

Input Script Syntax

This section describes the abstract syntax of an input script in EBNF. The convention is that $\langle \mathit{UpperCase} \rangle$ production names denote non-terminals, and $\langle \mathit{lowercase} \rangle$ names denote lexemes.

There are the following lexical classes (see Section 2.1): $\langle \mathit{constant} \rangle$ and $\langle \mathit{typename} \rangle$ are **Constants**, $\langle \mathit{label} \rangle$ and $\langle \mathit{varid} \rangle$ are **Variables**, and $\langle \mathit{muvar} \rangle$ are **Relational variables**. The class $\langle \mathit{int} \rangle$ signifies integers.

A.1 Joint Protocols and the Environment

$\langle \mathit{JointProtocol} \rangle$ ::= $\langle \mathit{Environment} \rangle$ $\langle \mathit{Protocol} \rangle$*

$\langle \mathit{Environment} \rangle$ ::= $\langle \mathit{EnvHeader} \rangle$
   $\langle \mathit{TypeDec} \rangle$* $\langle \mathit{EnvVarDec} \rangle$*
   $\langle \mathit{EnvAgentDec} \rangle$*
   $\langle \mathit{EnvInitCond} \rangle$?
   $\langle \mathit{EnvResolve} \rangle$?
   $\langle \mathit{EnvSpec} \rangle$+
   $\langle \mathit{EnvFairness} \rangle$*

$\langle \mathit{EnvHeader} \rangle$ ::= ‘`environment`’ string

$\langle \mathit{TypeDec} \rangle$ ::= ‘`type`’ $\langle \mathit{typename} \rangle$ ‘`=`’ ‘`{`’ $\langle \mathit{Enumeration} \rangle$ ‘`}`’

$\langle \mathit{Enumeration} \rangle$ ::= $\langle \mathit{Constant} \rangle$ (‘`,`’ $\langle \mathit{Constant} \rangle$)+ | $\langle \mathit{int} \rangle$ ‘`..`’ $\langle \mathit{int} \rangle$

$\langle \mathit{EnvVarDec} \rangle$ ::= $\langle \mathit{VarDec} \rangle$

$\langle \mathit{EnvAgentDec} \rangle$ ::= ‘`agent`’ $\langle \mathit{constant} \rangle$ $\langle \mathit{String} \rangle$ ‘`(`’ [$\langle \mathit{VarList} \rangle$] ‘`)`’

$\langle \mathit{VarList} \rangle$ ::= $\langle \mathit{Var} \rangle$ (‘`,`’ $\langle \mathit{Var} \rangle$)*

$\langle \mathit{EnvInitCond} \rangle$ ::= ‘`init_cond`’ ‘`=`’ $\langle \mathit{Expr} \rangle$

$\langle \mathit{EnvFairness} \rangle$ ::= ‘`fairness`’ ‘`=`’ KF

A.1.1 Environment Resolve-clause

$\langle \mathit{EnvResolve} \rangle$ ::= ‘`resolve`’ $\langle \mathit{ResolveBlock} \rangle$

$\langle \mathit{ResolveBlock} \rangle$ ::= ‘`begin`’ $\langle \mathit{ResolveStatement} \rangle$ (‘`;`’ $\langle \mathit{ResolveStatement} \rangle$)* ‘`end`’

⟨*ResolveStatement*⟩ ::= ⟨*ResolveBlock*⟩
  | ‘`if`’ ⟨*ResolveClause*⟩ (‘`[]`’ ⟨*ResolveClause*⟩)* [⟨*ResolveClauseOtherwise*⟩] ‘`fi`’
  | ‘`if`’ ⟨*ResolveExpr*⟩ ‘`then`’ ⟨*ResolveStatement*⟩ ‘`else`’ ⟨*ResolveStatement*⟩
  | ‘`skip`’
  | ⟨*varid*⟩ ‘`:=`’ ⟨*Expr*⟩

⟨*ResolveClause*⟩ ::= ⟨*ResolveExpr*⟩ ‘`->`’ ⟨*ResolveStatement*⟩

⟨*ResolveClauseOtherwise*⟩ ::= ‘`otherwise`’ ‘`->`’ ⟨*ResolveStatement*⟩

⟨*ResolveExpr*⟩ ::= ⟨*ResolveExpr*⟩

⟨*ResolveExpr*⟩ ::= ⟨*Var*⟩ | ⟨*Constant*⟩
  | ⟨*constant*⟩ ‘`.`’ ⟨*constant*⟩
  | ⟨*ResolveExpr*⟩ ⟨*BoolBinOp*⟩ ⟨*ResolveExpr*⟩
  | ‘`neg`’ ⟨*ResolveExpr*⟩ | ‘`(`’ ⟨*ResolveExpr*⟩ ‘`)`’

A.2 Agent Protocols

⟨*Protocol*⟩ ::= ⟨*ProtocolHeader*⟩ ⟨*EnvVars*⟩ ⟨*LocalVars*⟩ ⟨*Block*⟩

⟨*ProtocolHeader*⟩ ::= ‘`protocol`’ ⟨*String*⟩

⟨*EnvVars*⟩ ::= ‘`(`’ [⟨*VarDec*⟩ (‘`,`’ ⟨*VarDec*⟩)*] ‘`)`’

⟨*LocalVars*⟩ ::= [⟨*VarDec*⟩+ [⟨*LocalVarInitCond*⟩]]

⟨*LocalVarInitCond*⟩ ::= ‘`where`’ (‘`all_init`’ | Expr)

⟨*Block*⟩ ::= ‘`begin`’ ⟨*LabelledStatement*⟩ (‘`;`’ ⟨*LabelledStatement*⟩)* ‘`end`’

⟨*LabelledStatement*⟩ ::= [⟨*label*⟩ ‘`::`’] ⟨*Statement*⟩

⟨*Statement*⟩ ::= ⟨*Block*⟩
  | ‘`if`’ ⟨*Clause*⟩ (‘`[]`’ ⟨*Clause*⟩)* [‘`[]`’ ‘`otherwise`’ ‘`->`’ ⟨*LabelledStatement*⟩] ‘`fi`’
  | ‘`do`’ ⟨*Clause*⟩ (‘`[]`’ ⟨*Clause*⟩)* [‘`[]`’ ‘`otherwise`’ ‘`->`’ ⟨*LabelledStatement*⟩] [‘`[]`’ ‘`break`’ ‘`->`’ ⟨*LabelledStatement*⟩] ‘`od`’
  | ‘`if`’ ⟨*Expr*⟩ ‘`then`’ ⟨*LabelledStatement*⟩ ‘`else`’ ⟨*LabelledStatement*⟩
  | ‘`while`’ ⟨*Expr*⟩ ‘`do`’ ⟨*LabelledStatement*⟩
  | ‘`skip`’
  | ⟨*varid*⟩ ‘`:=`’ ⟨*Expr*⟩
  | ‘`<<`’ ⟨*Action*⟩ [‘`|`’ ⟨*Assignments*⟩] ‘`>>`’

⟨*Clause*⟩ ::= ⟨*Expr*⟩ ‘`->`’ ⟨*LabelledStatement*⟩

⟨*Action*⟩ ::= ⟨*constant*⟩
  | ⟨*Var*⟩ ‘`.`’ ‘`write`’ ‘`(`’ ⟨*Expr*⟩ ‘`)`’
  | ⟨*varid*⟩ ‘`:=`’ ⟨*Var*⟩ ‘`.`’ ‘`read`’ ‘`(`’ ‘`)`’

⟨*Assignments*⟩ ::= ⟨*Assignment*⟩ (‘`;`’ ⟨*Assignment*⟩)*

A.3 Expressions

⟨*Expr*⟩ ::= ⟨*Var*⟩ | ⟨*Constant*⟩
| ⟨*Var*⟩ '`==`' ⟨*Constant*⟩ | ⟨*Var*⟩ '`/=`' ⟨*Constant*⟩
| ⟨*Var*⟩ ⟨*RelOp*⟩ ⟨*Constant*⟩ | ⟨*Constant*⟩ ⟨*RelOp*⟩ ⟨*Var*⟩
| ⟨*Var*⟩ '`in`' ⟨*ConstantList*⟩
| ⟨*Var*⟩ ⟨*ArithOp*⟩ ⟨*int*⟩
| ⟨*Expr*⟩ ⟨*BoolBinOp*⟩ ⟨*Expr*⟩
| '`neg`' ⟨*Expr*⟩ | '`(`' Expr '`)`'

⟨*ArithOp*⟩ ::= '`+`' | '`-`'

⟨*RelOp*⟩ ::= '`<`' | '`<=`' | '`>`' | '`>=`'

A.4 Specifications

⟨*EnvSpec*⟩ ::= '`spec_clock`' '`=`' KFltl
| '`spec_obs`' '`=`' KF
| '`spec_obs_ltl`' '`=`' KFltl
| '`spec_pr`' '`=`' KFltl

KF is the CTL + mu-calculus + knowledge specification language.

⟨*KF*⟩ ::= Proposition
| ⟨*KF*⟩ ⟨*BoolBinOp*⟩ ⟨*KF*⟩
| '`neg`' ⟨*KF*⟩ | '`(`' ⟨*KF*⟩ '`)`'
| '`AX`' ⟨*KF*⟩ | '`EX`' ⟨*KF*⟩
| '`A[`' ⟨*KF*⟩ '`U`' ⟨*KF*⟩ '`]`' | '`E[`' ⟨*KF*⟩ '`U`' ⟨*KF*⟩ '`]`'
| '`AF`' ⟨*KF*⟩ | '`EF`' ⟨*KF*⟩
| '`AG`' ⟨*KF*⟩ | '`EG`' ⟨*KF*⟩
| ⟨*muvar*⟩
| '`gfp`' ⟨*muvar*⟩ ⟨*KF*⟩ | '`lfp`' ⟨*muvar*⟩ ⟨*KF*⟩
| '`Knows`' ⟨*constant*⟩ ⟨*KF*⟩
| '`CK`' ⟨*AgentList*⟩ ⟨*KF*⟩ | '`CK`' ⟨*KF*⟩

$KFltl$ is the LTL + knowledge specification language.

⟨*KFltl*⟩ ::= ⟨*Proposition*⟩
| ⟨*KFltl*⟩ ⟨*BoolBinOp*⟩ ⟨*KFltl*⟩
| '`neg`' ⟨*KFltl*⟩ | '`(`' ⟨*KFltl*⟩ '`)`'
| '`F`' ⟨*KFltl*⟩ | '`G`' ⟨*KFltl*⟩
| ⟨*KFltl*⟩ '`U`' ⟨*KFltl*⟩
| '`X`' ⟨*KFltl*⟩ | '`X`' ⟨*int*⟩ ⟨*KFltl*⟩
| '`Knows`' ⟨*constant*⟩ ⟨*KFltl*⟩
| '`CK`' ⟨*AgentList*⟩ ⟨*KFltl*⟩ | '`CK`' ⟨*KFltl*⟩

The basic propositions.

⟨*Proposition*⟩ ::= ⟨*QualifiedVar*⟩
| ⟨*Constant*⟩
| ⟨*QualifiedVar*⟩ '`==`' ⟨*Constant*⟩ | ⟨*QualifiedVar*⟩ '`/=`' ⟨*Constant*⟩
| ⟨*QualifiedVar*⟩ ⟨*RelOp*⟩ ⟨*Constant*⟩ | ⟨*Constant*⟩ ⟨*RelOp*⟩ ⟨*QualifiedVar*⟩
| ⟨*QualifiedVar*⟩ '`in`' ⟨*ConstantList*⟩
| ⟨*constant*⟩ '`.`' '`terminated`'

⟨*QualifiedVar*⟩ ::= [⟨*constant*⟩ ‘`.`’] Var

⟨*AgentList*⟩ ::= ‘`{`’ ⟨*constant*⟩ (‘`,`’ ⟨*constant*⟩)* ‘`}`’

⟨*LabelList*⟩ ::= ‘`{`’ ⟨*label*⟩ (‘`,`’ ⟨*label*⟩)* ‘`}`’

A.5 Common Productions

⟨*Assignment*⟩ ::= ⟨*Var*⟩ ‘`:=`’ ⟨*Expr*⟩

⟨*BoolBinOp*⟩ ::= ‘`/\`’ | ‘`\/`’ | ‘`->`’ | ‘`<->`’ | ‘`xor`’

⟨*ConstantList*⟩ ::= ‘`{`’ (⟨*Constant*⟩ ‘`..`’ ⟨*Constant*⟩ | ⟨*Constant*⟩ (‘`,`’ ⟨*Constant*⟩)+) ‘`}`’

⟨*Constant*⟩ ::= ⟨*int*⟩ | ⟨*constant*⟩

⟨*Type*⟩ ::= ⟨*TypeAttr*⟩* ⟨*TypeDesc*⟩

⟨*TypeAttr*⟩ ::= ‘`constant`’ | ‘`observable`’

⟨*TypeDesc*⟩ ::= ⟨*typename*⟩ | ⟨*typename*⟩ ‘`[`’ [⟨*int*⟩] ‘`]`’

⟨*Var*⟩ ::= ⟨*varid*⟩ | ⟨*varid*⟩ ‘`[`’ (‘`self`’ | ⟨*int*⟩) ‘`]`’

⟨*VarDec*⟩ ::= ⟨*varid*⟩ ‘`:`’ ⟨*Type*⟩

Appendix B

Operational Semantics

The motivation for providing a detailed operational semantics is to make the timing model precise.

There are two major constraints on the design of the language:

1. The agents need to have control over which action they emit at all times (see the Robot example in Section 4.1 for further details).
2. Automata must be naturally expressible.

Also, arbitrary nesting of constructs and a simple, regular timing model are very desirable.

In the following description, let:

- Var be the type of variables.
- Val be the type of values.
- $\rho :: Var \to Val$ be a global state (gives values to all variables).
- $\langle elt_1, ..., elt_n \rangle$ be sequences of arbitrary objects.
- $[x \mapsto val] :: Var \to Val \to (Var \to Val) \to (Var \to Val)$ be a state transfomer:

$$([x \mapsto val]\ \rho)\ y = \begin{cases} val & : \quad x == y \\ \rho\ y & : \quad \text{otherwise} \end{cases}$$

- $f \circ g$ be function composition: $(f \circ g)\ x \equiv f(g(x))$. A summation-style $\bigcirc_i$ is also used, where $\bigcirc_0 = \mathbf{id}$ and $\bigcirc_i = (\bigcirc_{i-1} tail) \circ head$, where $\langle head | tail \rangle$ are a list of composable functions.

In order to implement the select/resolve model (Chapter 1), the overall state transition relation is split into two sets of relations – agent-local and environment. Both rely on another relation to give meaning to expressions.

Expression Semantics

$\rho : Expr \hookrightarrow Val$ is the expression evaluation function, described in Section B.2.

Agent-local transitions

For each agent A, two relations are defined:

- $\rho : ProgramText \leadsto_A ProgramText'$ are agent-internal transitions, which become stuck upon encountering a manifest action.
- $\rho : ProgramText \to_A (Action, \langle Assignment \rangle, ProgramText')$ is the *selection* phase (determines what action the agent performs in this time step).

The idea is that each agent executes as much $ProgramText$ as possible until it is manifest which action is to be performed (at which point the $\rightsquigarrow_A$ relation gets stuck). The $\rightarrow_A$ relation takes care of detaching this action and tracking where the agent is up to in its protocol.

The $\rightarrow_n$ relation is the synchronous, lock-step aggregate of all the $\rightarrow_A$ agent relations.

The definitions of these relations are given in Section B.3.

Environment Transitions

- $[\![\cdot]\!]_{(\rho,\mathcal{A})}$ generates a state transformer from the resolve clause, given the current state and the actions the agents have performed.
- $\rho : \langle ProgramText_1, ..., ProgramText_n\rangle \Rightarrow \rho' : \langle ProgramText'_1, ..., ProgramText'_n\rangle$ is the overall single-step state-transformation relation.

The definitions of these relations are given in Section B.4.

B.1 Pre-processing

A pre-processing pass is used to expand derived forms and normalise the programs.

1. Append **while True do skip** to the original program P. This ensures all programs generate infinite runs.
2. Make all variable names unique by qualifying local variables with agent names.
3. Eliminate derived forms using the rules in Sections B.2 and 2.3.3.
4. For all **if ... fi** statements: if present, replace an **otherwise** $\rightarrow C$ branch with $\bigwedge_i \neg cond_i \rightarrow C$, or add a $\bigwedge_i \neg cond_i \rightarrow$ **skip** branch if it is absent.

   (Note this expansion also applies to a resolve clause, if present.)
5. For all **do ... od** statements: if the **break** branch is absent, add **break** $\rightarrow$ **skip**.

B.2 Expression Semantics

Assume the following:

- The notion of $var == val$ is primitive.
- The standard boolean algebraic identities hold: $f \vee g = \neg(\neg f \wedge \neg g)$, $f \rightarrow g = \neg f \vee g$, $f \leftrightarrow g = (f \rightarrow g) \wedge (g \rightarrow f)$, f xor $g = (\neg x \wedge y) \vee (x \wedge \neg y)$.
- The expression is well-typed.
- There is a map $type :: Identifier \rightarrow \langle Constant\rangle$ (taking variables to sequences of constants).
- There are two operators on sequences:

  $pred :: \langle a\rangle \rightarrow a \rightarrow a$, which returns the predecessor of an element in the sequence, or the element if it is the first; and

  $succ :: \langle a\rangle \rightarrow a \rightarrow a$, which returns the successor of an element in the sequence, or the element if it is the last.

  (These operators simply implement truncating arithmetic, as noted in Section 2.3.1.)
- Derived forms in the expression have been expanded using the following rules:
  - $[x\ > elt] =\ x$ **in** {elements of $type\ x$ to the right of elt}

– $[x\ \ >= elt] =\ x$ **in** $\{elt,$ elements of *type* x to the right of $elt\}$

– $[x\ \ < elt] =\ x$ **in** $\{$elements of *type* x to the left of $elt\}$

– $[x\ \ <= elt] =\ x$ **in** $\{elt,$ elements of *type* x to the left of $elt\}$

– $[x\ /= elt] =\ \neg(x == elt)$

The definition of the function $\hookrightarrow :: State \rightarrow Expr \rightarrow Val$:

$$
\begin{array}{l}
\rho : (f /\backslash g) \hookrightarrow (\rho : f \hookrightarrow \mathit{True}) \wedge (\rho : g \hookrightarrow \mathit{True}) \\
\rho : (\texttt{neg}\ f) \hookrightarrow \neg(\rho : f \hookrightarrow \mathit{True}) \\
\rho : \mathit{var} \hookrightarrow \rho\ \mathit{var} \\
\rho : \mathit{val} \hookrightarrow \mathit{val} \\
\rho : \mathit{var} \in \{\mathit{val}_1, ..., \mathit{val}_n\} \hookrightarrow \mathbf{let}\ \mathit{val} = \rho\ \mathit{var}\ \mathbf{in}\ \mathit{val} == \mathit{val}_1 \vee ... \vee \mathit{val} == \mathit{val}_n \\
\rho : \mathit{var} + i \hookrightarrow \mathbf{let}\ \mathit{val} = \rho\ \mathit{var}\ \mathbf{in}\ \mathit{succ}^i\ (\mathit{type}\ \mathit{var})\ \mathit{val} \\
\rho : \mathit{var} - i \hookrightarrow \mathbf{let}\ \mathit{val} = \rho\ \mathit{var}\ \mathbf{in}\ \mathit{pred}^i\ (\mathit{type}\ \mathit{var})\ \mathit{val}
\end{array}
$$

B.3 Agent Protocol Semantics

The definition of $\rho : ProgramText \rightsquigarrow_A RestOfProgramText$:

$$
\textbf{Sequencing:} \quad \frac{\rho : C_1 \rightsquigarrow_A C_1'}{\rho : C_1; C_2 \rightsquigarrow_A C_1'; C_2}
$$

If: For each alternative i:

$$
\frac{\rho : \mathit{cond}_i \hookrightarrow True}{\rho : \mathbf{if}\ ...\ \mathit{cond}_i\ \rightarrow C_i\ ...\ \mathbf{fi} \rightsquigarrow_A C_i}
$$

Do: For each alternative i:

$$
\frac{\rho : \mathit{cond}_i \hookrightarrow True}{\rho : \mathbf{do}\ ...\ \mathit{cond}_i\ \rightarrow C_i\ ...\ \mathbf{od} \rightsquigarrow_A C_i; \mathbf{do}\ ...\ \mathbf{od}}
$$

The $\rho : ProgramText \rightsquigarrow_A RestOfProgramText$ relation is the reflexive, transitive closure of the above definitions. This relation is purposefully non-deterministic.

The definition of $\rho : ProgramText \rightarrow_A (\mathit{Action}, \langle \mathit{Assignment} \rangle, ProgramText')$:

$$
\frac{\rho : ProgramText \rightsquigarrow_A < \mathit{Action} | \langle \mathit{Assignment} \rangle >; ProgramText'}{\rho : ProgramText \rightarrow_A (\mathit{Action}, \langle \mathit{Assignment} \rangle, ProgramText')}
$$

We expect $\rho : C \rightsquigarrow_A C'$ to terminate for all C, and the overall agent relation $\rho : ProgramText \rightarrow_A (\mathit{Action}, \langle \mathit{Assignment} \rangle, ProgramText')$ to not get stuck.

Define:

$$
\begin{array}{c}
\rho : (ProgramText_1, ..., ProgramText_n) \rightarrow_n \\
((\mathit{Action}_1, \langle \mathit{Assignment} \rangle_1, ProgramText_1'), ..., (\mathit{Action}_n, \langle \mathit{Assignment} \rangle_n, ProgramText_n'))
\end{array}
$$

to be the composite of the individual transitions

$$
\rho : ProgramText_A \rightarrow_A (\mathit{Action}_A, \langle \mathit{Assignment} \rangle_A, ProgramText_A')
$$

for each agent $A \in \{1...n\}$. This relation is used directly by the resolution relation.

Note on why 'do' has a 'break' branch

As noted in Section 2.3.3, the protocol language is non-standard in that it allows the programmer to specify what happens immediately after all guards evaluate to false in a **do** construct. (In Dijkstra's version [3], there is always an implicit **skip** before the programmer regains control.) The reason we require continuous control was set out in Section 4.1.

A superficially plausible alternative semantics is to try to determine an action which will be enabled once the **do** loop terminates, and execute that. Unfortunately, this leads to problems. Take, for example, the following program:

```
do True ->
  do False -> <Action>
  od
od
```

under these semantics. In this case, there simply is no action that can be executed between the evaluation of the two guards, and so the $\rightsquigarrow_A$ relation must be non-terminating.

Given our constraint that guards should be instantaneously evaluated, the most natural thing to do is to add the **break** branch. An alternative solution would be to ban nested looping constructs, or require that the first statement in the body of a **do** statement always produce an action (i.e. be a non-looping construct).

B.4 Environment Resolve-clause Semantics

The environment's resolution protocol is expressed as a non-looping program in a subset of the language used to represent protocols. The most significant departure is that guards can mention agent-qualified actions.

As mentioned in Section 2.3.3, the system currently provides shared-variable **read()** and **write()** actions as primitives, distinct from the mechanism described here. A semantics for these primitives is omitted.

The definition of $[\![\cdot]\!]_{(\rho,\mathcal{A})} :: Statement \rightarrow (State, \{Action\}) \rightarrow (Var \rightarrow Val) \rightarrow (Var \rightarrow Val)$, giving meaning to a resolve clause in terms of state transformers:

$$\frac{\rho' = [\![C_1]\!]_{(\rho,\mathcal{A})}\ \rho}{[\![C_1; C_2]\!]_{(\rho,\mathcal{A})} = [\![C_2]\!]_{(\rho',\mathcal{A})} \circ [\![C_1]\!]_{(\rho,\mathcal{A})}}$$

For each alternative i:

$$\frac{(\rho,\mathcal{A}) \models cond_i}{[\![\mathbf{if}\ \ldots\ cond_i\ \rightarrow C_i\ \ldots\ \mathbf{fi}]\!]_{(\rho,\mathcal{A})} = [\![C_i]\!]_{(\rho,\mathcal{A})}}$$

$$[\![\mathbf{skip}]\!]_{(\rho,\mathcal{A})} = \mathbf{id}_{(\mathbf{Var}\rightarrow\mathbf{Val})\rightarrow(\mathbf{Var}\rightarrow\mathbf{Val})}$$

$$\frac{\rho : expr \hookrightarrow val}{[\![x := expr]\!]_{(\rho,\mathcal{A})} = [x \mapsto val]}$$

where $\mathbf{id}_{(\mathbf{Var}\rightarrow\mathbf{Val})\rightarrow(\mathbf{Var}\rightarrow\mathbf{Val})}$ is the identity function for the specified type.

This function is purposefully non-deterministic.

Conditions are evaluated with respect to a set of actions $\mathcal{A}$ and a state ρ. The base cases are as follows:

$$\frac{Action \in \mathcal{A}}{(\rho,\mathcal{A}) \models Action}$$

$$\frac{\rho\ Var = True}{(\rho, \mathcal{A}) \models Var}$$

and the recursive cases (for the logical connectives) are similar to those in Section B.2.

The definition of $\rho : (ProgramText_1, ..., ProgramText_n) \Rightarrow \rho' : (ProgramText'_1, ..., ProgramText'_n)$, taking states and program counters in one state to the next:

$$\frac{\begin{array}{c} \rho : (ProgramText_1, ..., ProgramText_n) \rightarrow_n \\ ((Action_1, \langle Assignment\rangle_1, ProgramText'_1), ..., (Action_n, \langle Assignment\rangle_n, ProgramText'_n)) \\ st_{env} = [\![ResolveClause]\!]_{(\rho,\mathcal{A})} \\ \text{for each agent } i\text{: } st_i = [\![Assignment_{(i,1)}]\!]_{(\rho,\mathcal{A})} \circ ... \circ [\![Assignment_{(i,n)}]\!]_{(\rho,\mathcal{A})}. \\ \rho' = (st_{env} \circ (\bigcirc_i\ st_i))\ \rho \end{array}}{\rho : (ProgramText_1, ..., ProgramText_n) \Rightarrow \rho' : (ProgramText'_1, ..., ProgramText'_n)}$$

Notes:

- The order of composing the Environment's and Agents' state transformers st_i doesn't matter as their domains are disjoint.
- Where there are several assignments in a single action statement, the variables in their right-hand-sides refer to the current state. For example, in a state where $\{x \mapsto True, y \mapsto False\}$, the action $< NillAction|x := y; y := x >$ gives rise to a state where $\{x \mapsto False, y \mapsto True\}$.

B.4.1 Runs

A *run* is a sequence of global states ρ_i conformant with the overall one-step transition relation $\Rightarrow$:

- The initial state of the system is $\rho^0 : (ProgramText_1, ..., ProgramText_n)$, where ρ^0 must satisfy all local and global initial constraints, and $ProgramText_i$ is the entire program for agent i.
- For each $i \geq 0$, we have (with superscripts indicating position in the run):

  $$\rho^i : (ProgramText^i_1, ..., ProgramText^i_n) \Rightarrow \rho^{i+1} : (ProgramText^{i+1}_1, ..., ProgramText^{i+1}_n)$$